**Ahmad Sibai - U22845213**

**Arjan Patel – U39273630**

**Tilak Patel - U32922919**

**8 Puzzle Problem Implemented with BFS, DFS, and Dijkstra’s Algorithms**

**Abstract**

The 8 puzzle is a game consisting of a 3 by 3 grid which contains numbers 0 – 8. The goal state has 1 2 3 in the first row, 8 0/blank 4 in the second row, and 7 6 5 in the final state. The objective of the game is to find the cheapest solution where you begin at a certain initial state and attempt to reach the goal state by moving the tiles. The solution is the cost which differs amongst different algorithms as you will see. A tile can move up, down, left, or right and the cost is $1 for algorithms BFS and DFS while the cost in Dijkstra’s is the same as the number of the tiles so moving tile 1 would cost $1 and moving tile 3 would cost $3. But before attempting to solve the initial state, you must also find out whether the initial state is solvable which depends on the number of inversions which is how many tiles come before other smaller tiles. The initial state must be arranged in a flat array such as [0, 1, 2, 5, 8, 7, 6, 3, 4] where each of these numbers in the array represents the elements in the grid. So, the first 3 numbers are all from the first row in order. 5 is the last element in the second row. Then we have 8, 7, 6 which are the 3 numbers in the last row of the grid but in backwards order. Finally, we have 3 and 4 which are the first two elements in the second row. After applying this to the goal state, you will see that it has 0 inversions. For other initial states, if the number of inversions is even, then that state can be solved by one of the algorithms. If the number of inversions is odd, then it cannot be solved, and the output of each code will be unsolvable.

**BFS Implementation**

Breadth-first search is an algorithm that is used to search a tree or graph data structure for a node that meets a certain set of criteria. My task was to use BFS to solve the 8-puzzle problem. A quick overview of how I solved the puzzle is first, I calculated the number of inversions in the input puzzle. If the puzzle has an even number of inversions that means it is solvable. If the puzzle has an odd number of inversions, it means the puzzle is not solvable. After, I implemented a priority queue to keep track of the states visited and explored. Then, I called the solve function which will use the priority queue and return the goal state with the cost of each move.

For the test cases given in the Final Project PDF, the first case had 13 inversions which is an odd number so we can’t solve the 8 puzzle. The second and third test cases had 2 and 4 inversions which is even so both of these puzzles are solvable. I obtained a cost of $4 for test case two and $8 for test case three. I observed that since the first test case was not solvable I got a run time of 0.006 seconds which is very quick because we didn’t need to run through the rest of the algorithm. For test case 2 I obtained a run time of 0.013 seconds with 35 states explored. In test case three I obtained a run time of 0.466 seconds with 329 states explored. I found this interesting because I realized that the number of states that were being explored that a profound effect on the run time. In conclusion, the more states that were explored, the larger the run time would be.

**DFS Implementation**

To use DFS to solve the 8 puzzle problem I will be using a search tree to search for the goal state and a stack to keep track of visited nodes/children. Now DFS is the worst out of the three when it comes to the cost of the path to reaching the goal state because it does not find the cheapest solution, and rather just a solution to the 8 puzzle problem.

In my code, I created a class Node which will represent a node in the tree. Before you run the code, you must have your initial state in the input.txt file in a 3 by 3 2D array format. Once the code runs, it will copy this into the initial\_state array which is also 3 by 3 to print that out. In addition, we will flatten out the array using [0, 1, 2, 5, 8, 7, 6, 3, 4] where the number represents the number in the 3 by 3 grid. So, the first 3 numbers are all from the first row in order. 5 is the last element in the second row. Then we have 8, 7, 6 which are the 3 numbers in the last row of the grid but in backwards order. Finally, we have 3 and 4 which are the first two elements in the second row. After this we check for inversions which is when we have an element and another smaller element after it. If this number of inversions is even, then we can proceed with finding a solution as the problem is solvable. If not, it is odd, and therefore deemed unsolvable. Because the goal state number of inversions, we can also compare that to the future states we encounter once we search through the tree because if those states also have 0 inversions then it must be a match and therefore mean we have reached a goal state.

The count\_inversions function counts the number of inversions using the method I described above. After this, we use modulus division in is\_solvable to determine if it is solvable. If it is, then we carry on and start forming the tree using find\_blank which finds the 0/blank on the board and get\_children which adds all the possible children to the parent node. Finally, dfs uses a stack to keep track of visited children and continues working until it reaches the goal state. It then outputs the goal state in addition to the cost of the path it took to get to the goal state from the root all the way to that node with the goal state.

**Dijkstra’s Implementation**

Djikstra’s Algorithm is an algorithm to find the shortest paths from the source to other vertices in a graph. There are a variety of different multiple data structures to implement the algorithm. This report discusses the program's functionality to solve the 8-puzzle problem and its time and space complexity.

Dijkstra’s Algorithm is a greedy algorithm that finds the minimum cost from one vertex to another and follows the idea of selecting the best available option at that time. To efficiently implement Dijkstra's Algorithm, we utilized a heap queue. The queue utilizes a min heap and the priority of each of the elements is determined by the attribute within the class Puzzle called “actual\_cost.” The elements are then compared within the \_\_it\_\_ method with the specific condition of ‘self\_actual\_cost < other.actual\_cost.’ This condition means that the element with the smallest cost will be at the top of the heap tree and the first one will pop out.

The get\_neighbors function in the class generates all possible moves neighboring states for the current state in the puzzle. From the index of the state, it finds all the possible moves and has all the moves programmed through to represent moving a tile in all directions. After finding a sequence of moves, the function returns a tuple containing the total cost of the solution. This is done by adding all the values of the valid moves. If no solution is found, then the function returns ‘None’.

The read\_puzzle\_from\_file and write\_puzzle\_to\_file functions are important for the implementation of the program as well. The first function reads in the matrix from a text file and then is used in the class puzzle through the state parameter. Following the read function, the write function writes the goal state into a text file based on the solution found in the solve\_puzzle function.

The is\_solvable function calculates the number of inversions from the initial state obtained through the read\_puzzle\_from\_file function. If the number of inversions is even, then the puzzle is solvable, and the program continues the process of solving the puzzle. If the number of inversions is odd, then the program will not solve the problem as it is unsolvable.

Finally, the solve\_puzzle function implements Dijkstra’s algorithm to solve the puzzle from the given initial state to the goal state of (123,804,765). The function returns a tuple of the total cost function of the solution and the list of moves that occurred for the function to reach the goal state. This function completes the list of moves by finding the cheapest solution of all the known possibilities of the puzzle.